1 THE FUNDAMENTALS OF TESTING

Peter Morgan

BACKGROUND

If you were buying a new car, you would not expect to take delivery from the showroom with a scratch down the side of the vehicle. The car should have five wheels, a steering wheel, an engine and all the other essential components, and it should come with appropriate documentation, with all pre-sales checks completed and passed satisfactorily. The car you receive should be the car described in the sales literature; it should have the correct engine size, the correct colour scheme, and whatever extras you have ordered, and performance in areas such as fuel consumption and maximum speed should match published figures. In short, a level of expectation is set by brochures, by your experience of sitting in the driving seat, and probably by a test drive. If your expectations are not met you will feel justifiably aggrieved.

This kind of expectation seems not to apply to new software installations; examples of software being delivered not working as expected, or not working at all, are common. Why is this? There is no single cause that can be rectified to solve the problem, but one important contributing factor is the inadequacy of the testing to which software applications are exposed.

Software testing is neither complex nor difficult to implement, yet it is a discipline that is seldom applied with anything approaching the necessary rigour to provide confidence in delivered software. Software testing is costly in human effort or in the technology that can multiply the effect of human effort, yet is seldom implemented at a level that will provide any assurance that software will operate effectively, efficiently or even correctly.

This book explores the fundamentals of this important but neglected discipline to provide a basis on which a practical and cost-effective software testing regime can be constructed.

INTRODUCTION

In this opening chapter we have three very important objectives to achieve. First, we will introduce you to the fundamental ideas that underpin the discipline of

software testing, and this will involve the use and explanation of some new terminology. Secondly, we will establish the structure that we have used throughout the book to help you to use the book as a learning and revision aid. Thirdly, we will use this chapter to point forward to the content of later chapters.

We begin by defining what we expect you to get from reading this chapter. The learning objectives below are based on those defined in the Software Foundation Certificate syllabus (ISTQB, 2010), so you need to ensure that you have achieved all of these objectives before attempting the examination.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions at the start of the chapter, the 'Check of understanding' boxes distributed throughout the text, and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers see the Introduction.

Why is testing necessary? (K2)

- Describe, with examples, the way in which a defect in software can cause harm to a person, to the environment or to a company.
- Distinguish between the root cause of a defect and its effects.
- Give reasons why testing is necessary by giving examples.
- Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality.
- Recall the terms error, defect, fault, failure and the corresponding terms mistake and bug. (K1)

What is testing? (K2)

- Recall the common objectives of testing. (K1)
- Provide examples for the objectives of testing in different phases of the software life cycle.
- Differentiate testing from debugging.

General testing principles (K2)

- Explain the fundamental principles in testing.

Fundamental test process (K1)

- Recall the five fundamental test activities and respective tasks from planning to test closure.

The psychology of testing (K2)

- Recall the psychological factors that influence the success of testing. (K1)
- Contrast the mindset of a tester and of a developer.

Self-assessment questions
The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are given at the end of the chapter.

Question SA1 (K1)
A bug or defect is:

a. a mistake made by a person;
b. a run-time problem experienced by a user;
c. the result of an error or mistake;
d. the result of a failure, which may lead to an error?

Question SA2 (K1)
The effect of testing is to:

a. increase software quality;
b. give an indication of the software quality;
c. enable those responsible for software failures to be identified;
d. show there are no problems remaining?

Question SA3 (K1)
What is retesting?

a. Running the same test again in the same circumstances to reproduce the problem.
b. A cursory run through a test pack to see if any new errors have been introduced.
c. Checking that the predetermined exit criteria for the test phase have been met.
d. Running a previously failed test against new software/data/documents to see if the problem is solved.

WHY SOFTWARE FAILS

Examples of software failure are depressingly common. Here are some you may recognise:

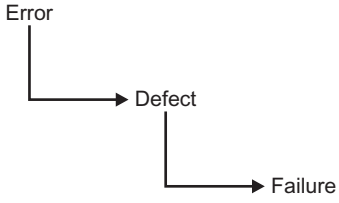
- The first launch of the European Space Agency Ariane 5 rocket in June 1996 failed after 37½ seconds. A software error caused the rocket to deviate from its vertical ascent, and the self-destruct capabilities were enacted before the then unpredictable flight path resulted in a bigger problem.
- When the UK Government introduced online filing of tax returns, a user could sometimes see the amount that a previous user earned. This was regardless of the physical location of the two applicants.
- In November 2005, information on the UK's top 10 wanted criminals was displayed on a website. The publication of this information was described in newspapers and on morning radio and television and, as a result, many people attempted to access the site. The performance of the website proved inadequate under this load and the website had to be taken offline. The publicity created performance peaks beyond the capacity of the website.
- When a well-known online book retailer first went live, ordering a negative number of books meant that the transaction sum involved was refunded to the 'purchaser'. Development staff had not anticipated that anyone would attempt to purchase a negative number of books. Code was developed to allow refunds to customers to be made by administrative staff – but self-requested refunds are not valid.
- A small, one-line, change in the billing system of an electrical provider blacked out the whole of a major US city.

What is it about these examples that make them so startling? Is it a sense that something fairly obvious was missed? Is it the feeling that, expensive and important as they were, the systems were allowed to enter service before they were ready? Do you think these systems were adequately tested? Obviously they were not, but in this book we want to explore why this was the case and why these kinds of failure continue to plague us.

To understand what is going on we need to start at the beginning, with the people who design systems. Do they make mistakes? Of course they do. People make mistakes because they are fallible, but there are also many pressures that make mistakes more likely. Pressures such as deadlines, complexity of systems and organisations, and changing technology all bear down on designers of systems and increase the likelihood of errors in specifications, in designs and in software code. These errors are where major system failures usually begin. If a document with an error in it is used to specify a component the component will be faulty and will probably exhibit incorrect behaviour. If this faulty component is built into a system the system may fail. While failure is not always guaranteed, it is likely that errors in specifications will lead to faulty components and faulty components will cause system failure.

An error (or mistake) leads to a defect, which can cause an observed failure (Figure 1.1).

Figure 1.1 Effect of an error



There are other reasons why systems fail. Environmental conditions such as the presence of radiation, magnetism, electronic fields or pollution can affect the operation of hardware and firmware and lead to system failure.

If we want to avoid failure we must either avoid errors and faults or find them and rectify them. Testing can contribute to both avoidance and rectification, as we will see when we have looked at the testing process in a little more detail. One thing is clear: if we wish to influence errors with testing we need to begin testing as soon as we begin making errors – right at the beginning of the development process – and we need to continue testing until we are confident that there will be no serious system failures – right at the end of the development process.

Before we move on, let us just remind ourselves of the importance of what we are considering. Incorrect software can harm:

- people (e.g. by causing an aircraft crash in which people die, or by causing a hospital life support system to fail);
- companies (e.g. by causing incorrect billing, which results in the company losing money);
- the environment (e.g. by releasing chemicals or radiation into the atmosphere).

Software failures can sometimes cause all three of these at once. The scenario of a train carrying nuclear waste being involved in a crash has been explored to help build public confidence in the safety of transporting nuclear waste by train. A failure of the train's on-board systems or of the signalling system that controls the train's movements could lead to catastrophic results. This may not be likely (we hope it is not) but it is a possibility that could be linked with software failure. Software failures, then, can lead to:

- Loss of money
- Loss of time

- Loss of business reputation
- Injury
- Death

KEEPING SOFTWARE UNDER CONTROL

With all of the examples we have seen so far, what common themes can we identify? There may be several themes that we could draw out of the examples, but one theme is clear: either insufficient testing or the wrong type of testing was done. More and better software testing seems a reasonable aim, but that aim is not quite as simple to achieve as we might expect.

Exhaustive testing of complex systems is not possible

With the Ariane 5 rocket launch, a particular software module was reused from the Ariane 4 programme. Only part of the functionality of the module was required, but the module was incorporated without changes. The unused functionality of the reused module indirectly caused a directional nozzle to move in an uncontrolled way because certain variables were incorrectly updated. In an Ariane 4 rocket the module would have performed as required, but in the Ariane 5 environment this malfunction in an area of software not even in use caused a catastrophic failure. The failure is well documented, but what is clear is that conditions were encountered in the first few seconds after the launch that were not expected, and therefore had not been tested.

If every possible test had been run, the problem would have been detected. However, if every test had been run, the testing would still be running now, and the ill-fated launch would never have taken place; this illustrates one of the general principles of software testing, which are explained below. With large and complex systems it will never be possible to test everything exhaustively; in fact it is impossible to test even moderately complex systems exhaustively.

In the Ariane 5 case it would be unhelpful to say that not enough testing was done; for this particular project, and for many others of similar complexity, that would certainly always be the case. In the Ariane 5 case the problem was that the right sort of testing was not done because the problem had not been detected.

Testing and risk

Risk is inherent in all software development. The system may not work or the project to build it may not be completed on time, for example. These uncertainties become more significant as the system complexity and the implications of failure increase. Intuitively, we would expect to test an automatic flight control system more than we would test a video game system. Why? Because the risk is greater. There is a greater probability of failure in the

more complex system and the impact of failure is also greater. What we test, and how much we test it, must be related in some way to the risk. Greater risk implies more and better testing.

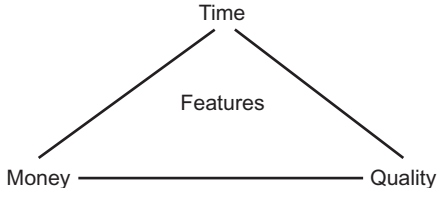
There is much more on risk and risk management in Chapter 5.

Testing and quality

Quality is notoriously hard to define. If a system meets its users' requirements that constitutes a good starting point. In the examples we looked at earlier the online tax returns system had an obvious functional weakness in allowing one user to view another user's details. While the user community for such a system is potentially large and disparate, it is hard to imagine any user that would find that situation anything other than unacceptable. In the top 10 criminals example the problem was slightly different. There was no failure of functionality in this case; the system was simply swamped by requests for access. This is an example of a non-functional failure, in that the system was not able to deliver its services to its users because it was not designed to handle the peak load that materialised after radio and TV coverage.

Of course the software development process, like any other, must balance competing demands for resources. If we need to deliver a system faster (i.e. in less time), for example, it will usually cost more. The items at the corners (or vertices) of the triangle of resources in Figure 1.2 are time, money and quality. These three affect one another, and also influence the features that are or are not included in the delivered software.

Figure 1.2 Resources triangle



One role for testing is to ensure that key functional and non-functional requirements are examined before the system enters service and any defects are reported to the development team for rectification. Testing cannot directly remove defects, nor can it directly enhance quality. By reporting defects it makes their removal possible and so contributes to the enhanced quality of the system. In addition, the systematic coverage of a software product in testing allows at least some aspects of the quality of the software to be measured. Testing is one component in the overall quality assurance activity that seeks to ensure that systems enter service without defects that can lead to serious failures.

Deciding when 'enough is enough'

How much testing is enough, and how do we decide when to stop testing?

We have so far decided that we cannot test everything, even if we would wish to. We also know that every system is subject to risk of one kind or another and that there is a level of quality that is acceptable for a given system. These are the factors we will use to decide how much testing to do.

The most important aspect of achieving an acceptable result from a finite and limited amount of testing is prioritisation. Do the most important tests first so that at any time you can be certain that the tests that have been done are more important than the ones still to be done. Even if the testing activity is cut in half it will still be true that the most important testing has been done. The most important tests will be those that test the most important aspects of the system: they will test the most important functions as defined by the users or sponsors of the system, and the most important non-functional behaviour, and they will address the most significant risks.

The next most important aspect is setting criteria that will give you an objective test of whether it is safe to stop testing, so that time and all the other pressures do not confuse the outcome. These criteria, usually known as completion criteria, set the standards for the testing activity by defining areas such as how much of the software is to be tested (this is covered in more detail in Chapter 4) and what levels of defects can be tolerated in a delivered product (which is covered in more detail in Chapter 5).

Priorities and completion criteria provide a basis for planning (which will be covered in Chapter 2 and Chapter 5) but the triangle of resources in Figure 1.2 still applies. In the end, the desired level of quality and risk may have to be compromised, but our approach ensures that we can still determine how much testing is required to achieve the agreed levels and we can still be certain that any reduction in the time or effort available for testing will not affect the balance – the most important tests will still be those that have already been done whenever we stop.

CHECK OF UNDERSTANDING

(1) Describe the interaction between errors, defects and failures.

(2) Software failures can cause losses. Give three consequences of software failures.

(3) What are the vertices of the 'triangle of resources'?

WHAT TESTING IS AND WHAT TESTING DOES

So far we have worked with an intuitive idea of what testing is. We have recognised that it is an activity used to reduce risk and improve quality by finding defects, which is all true. However, we need to understand a little

more about how software testing works in practice before we can think about how to implement effective testing.

Testing and debugging

Testing and debugging are different kinds of activity, both of which are very important. Debugging is the process that developers go through to identify the cause of bugs or defects in code and undertake corrections. Ideally some check of the correction is made, but this may not extend to checking that other areas of the system have not been inadvertently affected by the correction. Testing, on the other hand, is a systematic exploration of a component or system with the main aim of finding and reporting defects. Testing does not include correction of defects – these are passed on to the developer to correct. Testing does, however, ensure that changes and corrections are checked for their effect on other parts of the component or system.

Effective debugging is essential before testing begins to raise the level of quality of the component or system to a level that is worth testing, i.e. a level that is sufficiently robust to enable rigorous testing to be performed. Debugging does not give confidence that the component or system meets its requirements completely. Testing makes a rigorous examination of the behaviour of a component or system and reports all defects found for the development team to correct. Testing then repeats enough tests to ensure that defect corrections have been effective. So both are needed to achieve a quality result.

Static testing and dynamic testing

Static testing is the term used for testing where the code is not exercised. This may sound strange, but remember that failures often begin with a human error, namely a mistake in a document such as a specification. We need to test these because errors are much cheaper to fix than defects or failures (as you will see). That is why testing should start as early as possible, another basic principle explained in more detail later in this chapter. Static testing involves techniques such as reviews, which can be effective in preventing defects, e.g. by removing ambiguities and errors from specification documents; this is a topic in its own right and is covered in detail in Chapter 3. Dynamic testing is the kind that exercises the program under test with some test data, so we speak of test execution in this context. The discipline of software testing encompasses both static and dynamic testing.

Testing as a process

We have already seen that there is much more to testing than test execution. Before test execution there is some preparatory work to do to design the tests and set them up; after test execution there is some work needed to record the results and check whether the tests are complete. Even more important than this is deciding what we are trying to achieve with the testing and setting clear objectives for each test. A test designed to give confidence that a program functions according to its specification, for example, will be quite different from one designed to find as many defects as possible. We define a test process to ensure that we do not miss critical steps and that we do things in the right order. We will return to this important topic later, where we explain the fundamental test process in detail.

Testing as a set of techniques

The final challenge is to ensure that the testing we do is effective testing. It might seem paradoxical, but a good test is one that finds a defect if there is one present. A test that finds no defect has consumed resources but added no value; a test that finds a defect has created an opportunity to improve the quality of the product. How do we design tests that find defects? We actually do two things to maximise the effectiveness of the tests. First we use well-proven test design techniques, and a selection of the most important of these is explained in detail in Chapter 4. The techniques are all based on certain testing principles that have been discovered and documented over the years, and these principles are the second mechanism we use to ensure that tests are effective. Even when we cannot apply rigorous test design for some reason (such as time pressures) we can still apply the general principles to guide our testing. We turn to these next.

CHECK OF UNDERSTANDING

(1) Describe static testing and dynamic testing.

(2) What is debugging?

(3) What other elements apart from 'test execution' are included in 'testing'?

GENERAL TESTING PRINCIPLES

Testing is a very complex activity, and the software problems described earlier highlight that it can be difficult to do well. We now describe some general testing principles that help testers, principles that have been developed over the years from a variety of sources. These are not all obvious, but their purpose is to guide testers, and prevent the types of problems described previously. Testers use these principles with the test techniques described in Chapter 4.

Testing shows the presence of bugs

Running a test through a software system can only show that one or more defects exist. Testing cannot show that the software is error free. Consider whether the top 10 wanted criminals website was error free. There were no functional defects, yet the website failed. In this case the problem was non-functional and the absence of defects was not adequate as a criterion for release of the website into operation.

In Chapter 2, we will discuss retesting, when a previously failed test is rerun, to show that under the same conditions, the reported problem no longer exists. In this type of situation, testing can show that one particular problem no longer exists.

Although there may be other objectives, usually the main purpose of testing is to find defects. Therefore tests should be designed to find as many defects as possible.

Exhaustive testing is impossible

If testing finds problems, then surely you would expect more testing to find additional problems, until eventually we would have found them all. We discussed exhaustive testing earlier when looking at the Ariane 5 rocket launch, and

concluded that for large complex systems, exhaustive testing is not possible. However, could it be possible to test small pieces of software exhaustively, and only incorporate exhaustively tested code into large systems?

Exhaustive testing – a test approach in which all possible data combinations are used. This includes implicit data combinations present in the state of the software/data at the start of testing.

Consider a small piece of software where one can enter a password, specified to contain up to three characters, with no consecutive repeating entries. Using only western alphabetic capital letters and completing all three characters, there are $26 \times 26 \times 26$ input permutations (not all of which will be valid). However, with a standard keyboard, there are not $26 \times 26 \times 26$ permutations, but a much higher number, $256 \times 256 \times 256$, or 2^{24}. Even then, the number of possibilities is higher. What happens if three characters are entered, and the 'delete last character' right arrow key removes the last two? Are special key combinations accepted, or do they cause system actions (Ctrl + P, for example)? What about entering a character, and waiting 20 minutes before entering the other two characters? It may be the same combination of keystrokes, but the circumstances are different. We can also include the situation where the 20-minute break occurs over the change-of-day interval. It is not possible to say whether there are any defects until all possible input combinations have been tried.

Even in this small example, there are many, many possible data combinations to attempt.

Unless the application under test (AUT) has an extremely simple logical structure and limited input, it is not possible to test all possible combinations of data input and circumstances. For this reason, risk and priorities are used to concentrate on the most important aspects to test. Both 'risk' and 'priorities' are covered later in more detail. Their use is important to ensure that the most important parts are tested.

Early testing

When discussing why software fails, we briefly mentioned the idea of early testing. This principle is important because, as a proposed deployment date approaches, time pressure can increase dramatically. There is a real danger that testing will be squeezed, and this is bad news if the only testing we are doing is after all the development has been completed. The earlier the testing activity is started, the longer the elapsed time available. Testers do not have to wait until software is available to test.

Work-products are created throughout the software development life cycle (SDLC). As soon as these are ready, we can test them. In Chapter 2, we will see that requirement documents are the basis for acceptance testing, so the creation of acceptance tests can begin as soon as requirement documents are available. As we create these tests, it will highlight the contents of the requirements. Are individual requirements testable? Can we find ambiguous or missing requirements?

Many problems in software systems can be traced back to missing or incorrect requirements. We will look at this in more detail when we discuss reviews in Chapter 3. The use of reviews can break the 'error–defect–failure' cycle. In early testing we are trying to find errors and defects before they are passed to the next stage of the development process. Early testing techniques are attempting to show that what is produced as a system specification, for example, accurately reflects that which is in the requirement documents. Ed Kit (Kit, 1995) discusses identifying and eliminating errors at the part of the SDLC in which they are introduced. If an error/defect is introduced in the coding activity, it is preferable to detect and correct it at this stage. If a problem is not corrected at the stage in which it is introduced, this leads to what Kit calls 'errors of migration'. The result is rework. We need to rework not just the part where the mistake was made, but each subsequent part where the error has been replicated. A defect found at acceptance testing where the original mistake was in the requirements will require several work-products to be reworked, and subsequently to be retested.

Studies have been done on the cost impacts of errors at the different development stages. Whilst it is difficult to put figures on the relative costs of finding defects at different levels in the SDLC, Table 1.1 does concentrate the mind!

Table 1.1 Comparative cost to correct errors

Stage error is found	Comparative cost
Requirements	$1
Coding	$10
Program testing	$100
System testing	$1,000
User acceptance testing	$10,000
Live running	$100,000

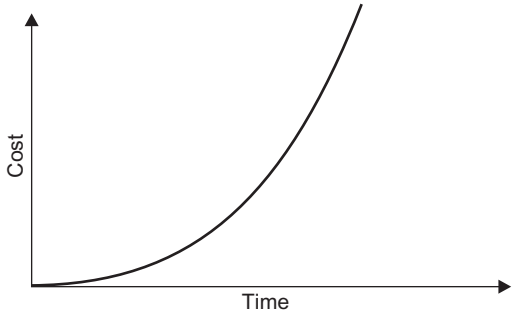
This is known as the cost escalation model.

What is undoubtedly true is that the graph of the relative cost of early and late identification/correction of defects rises very steeply (Figure 1.3).

The earlier a problem (defect) is found, the less it costs to fix.

The objectives of various stages of testing can be different. For example, in the review processes, we may focus on whether the documents are consistent and no errors have been introduced when the documents were produced. Other stages of testing can have other objectives. The important point is that testing has defined objectives.

Figure 1.3 Effect of identification time on cost of errors



Defect clustering

Problems do occur in software! It is a fact. Once testing has identified (most of) the defects in a particular application, it is at first surprising that the spread of defects is not uniform. In a large application, it is often a small number of modules that exhibit the majority of the problems. This can be for a variety of reasons, some of which are:

- System complexity.
- Volatile code.
- The effects of change upon change.
- Development staff experience.
- Development staff inexperience.

This is the application of the Pareto principle to software testing: approximately 80 per cent of the problems are found in about 20 per cent of the modules. It is useful if testing activity reflects this spread of defects, and targets areas of the application under test where a high proportion of defects can be found. However, it must be remembered that testing should not concentrate exclusively on these parts. There may be fewer defects in the remaining code, but testers still need to search diligently for them.

The pesticide paradox

Running the same set of tests continually will not continue to find new defects. Developers will soon know that the test team always tests the boundaries of conditions, for example, so they will test these conditions before the software is delivered. This does not make defects elsewhere in the code less likely, so continuing to use the same test set will result in decreasing effectiveness of the tests. Using other techniques will find different defects.

For example, a small change to software could be specifically tested and an additional set of tests performed, aimed at showing that no additional problems have been introduced (this is known as regression testing). However, the software may fail in production because the regression tests are no longer relevant to the

requirements of the system or the test objectives. Any regression test set needs to change to reflect business needs, and what are now seen as the most important risks. Regression testing will be discussed later in this chapter, but is covered in more detail in Chapter 2.

Testing is context dependent
Different testing is necessary in different circumstances. A website where information can merely be viewed will be tested in a different way to an e-commerce site, where goods can be bought using credit/debit cards. We need to test an air traffic control system with more rigour than an application for calculating the length of a mortgage.

Risk can be a large factor in determining the type of testing that is needed. The higher the possibility of losses, the more we need to invest in testing the software before it is implemented. A fuller discussion of risk is given in Chapter 5.

For an e-commerce site, we should concentrate on security aspects. Is it possible to bypass the use of passwords? Can 'payment' be made with an invalid credit card, by entering excessive data into the card number? Security testing is an example of a specialist area, not appropriate for all applications. Such types of testing may require specialist staff and software tools. Test tools are covered in more detail in Chapter 6.

Absence of errors fallacy
Software with no known errors is not necessarily ready to be shipped. Does the application under test match up to the users' expectations of it? The fact that no defects are outstanding is not a good reason to ship the software.

Before dynamic testing has begun, there are no defects reported against the code delivered. Does this mean that software that has not been tested (but has no outstanding defects against it) can be shipped? We think not!

CHECK OF UNDERSTANDING

(1) Why is 'zero defects' an insufficient guide to software quality?

(2) Give three reasons why defect clustering may exist.

(3) Briefly justify the idea of early testing.

FUNDAMENTAL TEST PROCESS

We previously determined that testing is a process, discussed above. This process is detailed in what has become known as the fundamental test process, a key element of what testers do, and is applicable at all stages of testing.

The most visible part of testing is running one or more tests: test execution. We also have to prepare for running tests, analyse the tests that have been run, and

see whether testing is complete. Both planning and analysing are very necessary activities that enhance and amplify the benefits of the test execution itself. It is no good testing without deciding how, when and what to test. Planning is also required for the less formal test approaches such as exploratory testing, covered in more detail in Chapter 4.

The fundamental test process consists of five parts that encompass all aspects of testing (Figure 1.4):

(1) Planning and control.

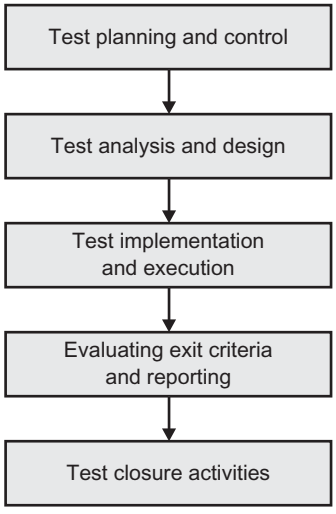
(2) Analysis and design.

(3) Implementation and execution.

(4) Evaluating exit criteria and reporting.

(5) Test closure activities.

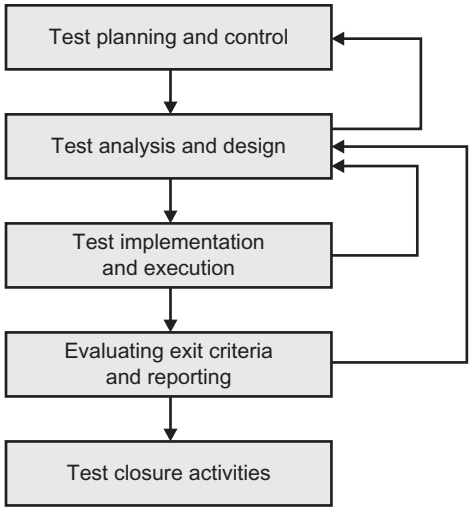
Figure 1.4 Fundamental test process



Although the main activities are in a broad sequence, they are not undertaken in a rigid way. An earlier activity may need to be revisited. A defect found in test execution can sometimes be resolved by adding functionality that was originally not present (either missing in error, or the new facilities are needed to make the other part correct). The new features themselves have to be tested, so even though implementation and execution are in progress, the 'earlier' activity of analysis and design has to be performed for the new features (Figure 1.5).

We sometimes need to do two or more of the main activities in parallel. Time pressure can mean that we begin test execution before all tests have been designed.

Figure 1.5 Iteration of activities



Test planning and control

Planning is determining what is going to be tested, and how this will be achieved. It is where we draw a map; how activities will be done; and who will do them. Test planning is also where we define the test completion criteria. Completion criteria are how we know when testing is finished. Control, on the other hand, is what we do when the activities do not match up with the plans. It is the ongoing activity where we compare the progress against the plan. As progress takes place, we may need to adjust plans to meet the targets, if this is possible. Therefore we need to undertake both planning and control throughout the testing activities. We plan at the outset, but as testing progresses, undertake monitoring and control activities (monitoring to measure what has happened, control to adjust future activities in the light of experience). Monitoring and control feed back into the continual activity of planning. The activities of planning and control are developed in more detail in Chapter 5.

Test analysis and design

Analysis and design are concerned with the fine detail of what to test (test conditions), and how to combine test conditions into test cases, so that a small number of test cases can cover as many of the test conditions as possible. The analysis and design stage is the bridge between planning and test execution. It is looking backward to the planning (schedules, people, what is going to be tested) and forward to the execution activity (test expected results, what environment will be needed).

A part of the design process needs to consider the test data that will be required for the test conditions and test cases that have been drawn up.

Test design involves predicting how the software under test should behave in a given set of circumstances. Sometimes the expected outcome of a test is

trivial: when ordering books from an online book retailer, for instance, under no circumstances should money be refunded to the customer's card without intervention from a supervisor. If we do not detail expected outcomes before starting test execution, there is a real danger that we will miss the one item of detail that is vital, but wrong.

These topics will be discussed in more detail in Chapter 4, when test case design techniques are presented. The main points of this activity are as follows:

- Reviewing requirements, architecture, design, interface specifications and other parts, which collectively comprise the test basis.
- Analysing test items, the specification, behaviour and structure to identify test conditions and test data required.
- Designing the tests, including assigning priority.
- Determining whether the requirements and the system are testable.
- Detailing what the test environment should look like, and whether there are any infrastructure and tools required.
- Highlighting the test data required for the test conditions and test cases.
- Creating bi-directional traceability between test basis and test cases.

Test implementation and execution

The test implementation and execution activity involves running tests, and this will include where necessary any set-up/tear-down activities for the testing. It will also involve checking the test environment before testing begins. Test execution is the most visible part of testing, but it is not possible without other parts of the fundamental test process. It is not just about running tests. As we have already mentioned, the most important tests need to be run first. How do we know what are the most important tests to run? This is determined during the planning stages, and refined as part of test design.

One important aspect undertaken at this stage is combining test cases into an overall run procedure, so that test time can be utilised efficiently. Here the logical ordering of tests is important so that, where possible, the outcome of one test creates the preconditions for one or more tests that are later in the execution sequence.

As tests are run, their outcome needs to be logged, and a comparison made between expected results and actual results. Whenever there is a discrepancy between the expected and actual results, this needs to be investigated. If necessary a test incident should be raised. Each incident requires investigation, although corrective action will not be necessary in every case. Test incidents will be discussed in Chapter 5.

When anything changes (software, data, installation procedures, user documentation, etc.), we need to do two kinds of testing on the software. First of all, tests should be run to make sure that the problem has been fixed. We also need to make sure that the changes have not broken the software elsewhere. These two types are usually called retesting and regression testing, respectively. In retesting we

are looking in fine detail at the changed area of functionality, whereas regression testing should cover all the main functions to ensure that no unintended changes have occurred. On a financial system, we should include end of day/end of month/end of year processing, for example, in a regression test pack.

Test implementation and execution is where the most visible test activities are undertaken, and usually have the following parts:

- Developing and prioritising test cases, creating test data, writing test procedures and, optionally, preparing test harnesses and writing automated test scripts.
- Collecting test cases into test suites, where tests can be run one after another for efficiency.
- Checking the test environment set-up is correct.
- Running test cases in the determined order. This can be manually or using test execution tools.
- Keeping a log of testing activities, including the outcome (pass/fail) and the versions of software, data, tools and testware (scripts, etc.).
- Comparing actual results with expected results.
- Reporting discrepancies as incidents with as much information as possible, including if possible causal analysis (code defect, incorrect test specification, test data error or test execution error).
- Where necessary, repeating test activities when changes have been made following incidents raised. This includes re-execution of a test that previously failed in order to confirm a fix (retesting), execution of a corrected test and execution of previously passed tests to check that defects have not been introduced (regression testing).

Evaluating exit criteria and reporting

Remember that exit criteria were defined during test planning and before test execution started. At the end of test execution, the test manager checks to see if these have been met. If the criterion was that there would be 85 per cent statement coverage (i.e. 85 per cent of all executable statements have been executed (see Chapter 4 for more detail)), and as a result of execution the figure is 75 per cent, there are two possible actions: change the exit criteria, or run more tests. It is possible that even if the preset criteria were met, more tests would be required. Also, writing a test summary for stakeholders would say what was planned, what was achieved, highlight any differences and in particular things that were not tested.

The fourth stage of the fundamental test process, evaluating exit criteria, comprises the following:

- Checking whether the previously determined exit criteria have been met.
- Determining if more tests are needed or if the specified exit criteria need amending.

- Writing up the result of the testing activities for the business sponsors and other stakeholders.

More detail is given on this subject in Chapter 5.

Test closure activities
Testing at this stage has finished. Test closure activities concentrate on making sure that everything is tidied away, reports written, defects closed, and those defects deferred for another phase clearly seen to be as such.

At the end of testing, the test closure stage is composed of the following:

- Ensuring that the documentation is in order; what has been delivered is defined (it may be more or less than originally planned), closing incidents and raising changes for future deliveries, documenting that the system has been accepted.
- Closing down and archiving the test environment, test infrastructure and testware used.
- Passing over testware to the maintenance team.
- Writing down the lessons learned from this testing project for the future, and incorporating lessons to improve the testing process ('testing maturity').

The detail of the test closure activities is discussed in Chapter 5.

CHECK OF UNDERSTANDING

(1) What are the stages in the fundamental test process (in the correct sequence)?

(2) Briefly compare regression testing and retesting.

(3) When should the expected outcome of a test be defined?

THE PSYCHOLOGY OF TESTING

A variety of different people may be involved in the total testing effort, and they may be drawn from a broad set of backgrounds. Some will be developers, some professional testers, and some will be specialists, such as those with performance testing skills, whilst others may be users drafted in to assist with acceptance testing. Who ever is involved in testing needs at least some understanding of the skills and techniques of testing to make an effective contribution to the overall testing effort.

Testing can be more effective if it is not undertaken by the individual(s) who wrote the code, for the simple reason that the creator of anything (whether it is software or a work of art) has a special relationship with the created object. The nature of that relationship is such that flaws in the created object are rendered invisible to the creator. For that reason it is important that someone other than the creator

should test the object. Of course we do want the developer who builds a component or system to debug it, and even to attempt to test it, but we accept that testing done by that individual cannot be assumed to be complete. Developers can test their own code, but it requires a mindset change, from that of a developer (to prove it works) to that of a tester (trying to show that it does not work). If there are separate individuals involved, there are no potential conflicts of interest.We therefore aim to have the software tested by someone who was not involved in the creation of the software; this approach is called test independence. Below are people who could test software, listed in order of increasing independence:

- Those who wrote the code.
- Members of the same development team.
- Members of a different group (independent test team).
- Members of a different company (a testing consultancy/outsourced).

Of course independence comes at a price; it is much more expensive to use a testing consultancy than to test a program oneself.

Testers and developers think in different ways. However, although we know that testers should be involved from the beginning, it is not always good to get testers involved in code execution at an early stage; there are advantages and disadvantages. Getting developers to test their own code has advantages (as soon as problems are discovered, they can be fixed, without the need for extensive error logs), but also difficulties (it is hard to find your own mistakes). People and projects have objectives, and we all modify actions to blend in with the goals. If a developer has a goal of producing acceptable software by certain dates, then any testing is aimed towards that goal.

If a defect is found in software, the software author may see this as criticism. Testers need to use tact and diplomacy when raising defect reports. Defect reports need to be raised against the software, not against the individual who made the mistake. The mistake may be in the code written, or in one of the documents upon which the code is based (requirement documents or system specification). When we raise defects in a constructive way, bad feeling can be avoided.

We all need to focus on good communication, and work on team building. Testers and developers are not opposed, but working together, with the joint target of better quality systems. Communication needs to be objective, and expressed in impersonal ways:

- The aim is to work together rather than be confrontational. Keep the focus on delivering a quality product.
- Results should be presented in a non-personal way. The work-product may be wrong, so say this in a non-personal way.
- Attempt to understand how others feel; it is possible to discuss problems and still leave all parties feeling positive.

- At the end of discussions, confirm that you have both understood and been understood. 'So, am I right in saying that your aim was to deliver on Friday by 12:00, even if you knew there were problems?'

As testers and developers, one of our goals is better quality systems delivered in a timely manner. Good communication between testers and the development teams is one way that this goal can be reached.

CHECK OF UNDERSTANDING

(1) When testing software, who has the highest level of independence?

(2) Contrast the advantages and disadvantages of developers testing their own code.

(3) Suggest three ways that confrontation can be avoided.

CODE OF ETHICS

One last topic that we need to address before we move onto the more detailed coverage of topics in the following chapters is that testers must adhere to a code of ethics: they are required to act in a professional manner. Testers can have access to confidential and/or privileged information, and they are to treat any information with care and attention, and act responsibly to the owner(s) of this information, employers and the wider public interest. Of course, anyone can test software, so the declaration of this code of conduct applies to those who have achieved software testing certification! The code of ethics applies to the following areas:

- Public – Certified software testers shall consider the wider public interest in their actions.
- Client and employer – Certified software testers shall act in the best interests of their client and employer (being consistent with the wider public interest).
- Product – Certified software testers shall ensure that the deliverables they provide (for any products and systems they work on) meet the highest professional standards possible.
- Judgement – Certified software testers shall maintain integrity and independence in their professional judgement.
- Management – Certified software test managers and leaders shall subscribe to and promote and ethical approach to the management of software testing.
- Profession – Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest.
- Colleagues – Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers.

- Self – Certified software testers shall participate in lifelong learning regarding the practice of their profession, and shall promote an ethical approach to the practice of the profession.

The code of ethics is far reaching in its aims, and a quick review of the eight points reveals interaction with other areas of the syllabus. The implementation of this code of ethics is expanded on in all chapters of this book, and perhaps is the reason for the whole book itself!

CHECK OF UNDERSTANDING

(1) Why do testers need to consider how they treat information they use?

(2) What eight areas should testers consider when conducting themselves in the workplace?

(3) What other sections of this chapter have described how testers should conduct themselves with colleagues?

SUMMARY

In this chapter, we have looked at key ideas that are used in testing, and introduced some terminology. We examined some of the types of software problems that can occur, and why the blanket explanation of 'insufficient testing' is unhelpful. The problems encountered then led us through some questions about the nature of testing, why errors and mistakes are made, and how these can be identified and eliminated. Individual examples enabled us to look at what testing can achieve, and the view that testing does not improve software quality, but provides information about that quality.

We have examined both general testing principles and a standard template for testing: the fundamental test process. These are useful and can be effective in identifying the types of problems we considered at the start of the chapter. The chapter finished by examining how developers and testers think, and looked at different levels of test independence, and how testers should behave by adhering to a code of ethics.

This chapter is an introduction to testing, and to themes that are developed later in the book. It is a chapter in its own right, but also points to information that will come later. A rereading of this chapter when you have worked through the rest of the book will place all the main topics into context.

REFERENCES

Kit, Edward (1995) *Software Testing in the Real World*. Addison-Wesley, Reading, MA.

Example examination questions with answers

E1. K1 question

Which of the following is correct?

Debugging is:

a. Testing/checking whether the software performs correctly.
b. Checking that a previously reported defect has been corrected.
c. Identifying the cause of a defect, repairing the code and checking the fix is correct.
d. Checking that no unintended consequences have occurred as a result of a fix.

E2. K2 question

Which of the following are aids to good communication, and which hinder it?

(i) Try to understand how the other person feels.
(ii) Communicate personal feelings, concentrating upon individuals.
(iii) Confirm the other person has understood what you have said and vice versa.
(iv) Emphasise the common goal of better quality.
(v) Each discussion is a battle to be won.

a. (i), (ii) and (iii) *aid*, (iv) and (v) *hinder*.
b. (iii), (iv) and (v) *aid*, (i) and (ii) *hinder*.
c. (i), (iii) and (iv) *aid*, (ii) and (v) *hinder*.
d. (ii), (iii) and (iv) *aid*, (i) and (v) *hinder*.

E3. K1 question

Which option is part of the 'implementation and execution' area of the fundamental test process?

a. Developing the tests.
b. Comparing actual and expected results.
c. Writing a test summary.
d. Analysing lessons learnt for future releases.

E4. K1 question

The five parts of the fundamental test process have a broad chronological order. Which of the options gives three different parts in the correct order?

a. Implementation and execution, planning and control, analysis and design.
b. Analysis and design, evaluating exit criteria and reporting, test closure activities.
c. Evaluating exit criteria and reporting, implementation and execution, analysis and design.
d. Evaluating exit criteria and reporting, test closure activities, analysis and design.

E5. K2 question
Which pair of definitions is correct?

a. Regression testing is checking that the reported defect has been fixed; retesting is testing that there are no additional problems in previously tested software.
b. Regression testing is checking there are no additional problems in previously tested software; retesting enables developers to isolate the problem.
c. Regression testing involves running all tests that have been run before; retesting runs new tests.
d. Regression testing is checking that there are no additional problems in previously tested software, retesting is demonstrating that the reported defect has been fixed.

E6. K1 question
Which statement is **most** true?

a. Different testing is needed depending upon the application.
b. All software is tested in the same way.
c. A technique that finds defects will always find defects.
d. A technique that has found no defects is not useful.

E7. K1 question
When is testing complete?

a. When time and budget are exhausted.
b. When there is enough information for sponsors to make an informed decision about release.
c. When there are no remaining high priority defects outstanding.
d. When every data combination has been exercised successfully.

E8. K1 question
Which list of levels of tester independence is in the correct order, starting with the **most** independent first?

a. Tests designed by the author; tests designed by another member of the development team; tests designed by someone from a different company.
b. Tests designed by someone from a different department within the company; tests designed by the author; tests designed by someone from a different company.
c. Tests designed by someone from a different company; tests designed by someone from a different department within the company; tests designed by another member of the development team.
d. Tests designed by someone from a different department within the company; tests designed by someone from a different company; tests designed by the author.

E9. K2 question

The following statements relate to activities that are part of the fundamental test process.

(i) Evaluating the testability of requirements.
(ii) Repeating testing activities after changes.
(iii) Designing the test environment set-up.
(iv) Developing and prioritising test cases.
(v) Verifying the environment is set up correctly.

Which statement below is TRUE?

a. (i) and (ii) are part of analysis and design, (iii), (iv) and (v) are part of test implementation and execution.
b. (i) and (iii) are part of analysis and design, (ii), (iv) and (v) are part of test implementation and execution.
c. (i) and (v) are part of analysis and design, (ii), (iii) and (iv) are part of test implementation and execution.
d. (i) and (iv) are part of analysis and design, (ii), (iii) and (v) are part of test implementation and execution.

E10. K2 question

Which statement correctly describes the **public** and **profession** aspects of the code of ethics?

a. **Public:** Certified software testers shall act in the best interests of their client and employer (being consistent with the wider public interest). **Profession:** Certified software testers shall advance the integrity and reputation of their industry consistent with the public interest.

b. **Public:** Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest. **Profession:** Certified software testers shall consider the wider public interest in their actions.

c. **Public:** Certified software testers shall consider the wider public interest in their actions. **Profession:** Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of their profession.

d. **Public:** Certified software testers shall consider the wider public interest in their actions. **Profession:** Certified software testers shall advance the integrity and reputation of their industry consistent with the public interest.

Answers to questions in the chapter

SA1. The correct answer is c.
SA2. The correct answer is b.
SA3. The correct answer is d.

Answers to example questions

E1. The correct answer is c.

a. is a brief definition of testing.
b. is retesting.
d. is regression testing.

E2. The correct answer is c.
If you are unsure why, revisit the section in this chapter on the psychology of testing.

E3. The correct answer is b.

a. is part of 'Analysis and design'.
c. is part of 'Evaluating exit criteria and reporting'.
d. is part of 'Test closure activities'.

E4. The correct answer is b.
All other answers have at least one stage of the fundamental test process in the wrong sequence.

E5. The correct answer is d.
Regression testing is testing that nothing has **regressed**. Retesting (or confirmation testing) **confirms** the fix is correct by running the same test after the fix has been made. No other option has both of these as true.

E6. The correct answer is a.
This is a restatement of the testing principle 'Testing is context dependent'.

E7. The correct answer is b.
Sometimes time/money does signify the end of testing, but it is really complete when everything that was set out in advance has been achieved.

E8. The correct answer is c.
This option has someone **nearer** to the written code in each statement. All other options are not in this order.

E9. The correct answer is b.
All other answers contain an activity identified as analysis and design that is part of implementation and test execution.

E10. The correct answer is d.
All the answers reflect the definition of **two** of the items from the code of ethics, and care must be taken in searching for the **Public** item because 'public' or

'public interest' are used in several of the eight items in the code. The key is that 'public' is the main item, rather than a subsidiary. In the order given in the options, a. reflects **Client and employer** and **Profession** while b. gives **Profession** and **Public** (the correct choices, but the wrong way round). Option c. gives **Public** and **Self**, leaving the last option d. to give **Public** and **Profession**.